Concurrent Enforcement of Usage Control Policies

Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan
Software Technology Research Laboratory,
De Montfort University, Leicester, UK LE1 9BH
Email: {heljanic, acau, fsiewe, hzedan}@dmu.ac.uk

Abstract— **Policy-based approaches to the management of systems distinguish between the specification of requirements, in the form of policies, and their enforcement on the system. In this work we focus on the latter aspect and investigate the enforcement of stateful policies in a concurrent environment. As a representative of stateful policies we use the UCON model and show how dependencies between policy rules affect their enforcement. We propose a technique for enforcing policies concurrently based on the static analysis of dependencies between policies. The potential of our technique for improving the efficacy of enforcement mechanisms is illustrated using a small, but representative example.**

I. Introduction

Policy-based approaches to the management of systems rely on the ability of administrators to define their requirements concisely in form of policies and on mechanisms that enforce these policies in the system. In the recent years much research effort has been placed on the development of policy languages, improving their expressiveness to capture complex requirements and tool-support for policy specification and analysis. However, work on the actual design of mechanisms that enforce policies in the system and their efficiency and correctness has been payed less attention. In this paper we show how the introduction of stateful policies, e.g. attribute-based policies, history-based policies or those that can express dynamic separation of duty requirements, impacts on the design and implementation of enforcement mechanisms. We highlight potential problems that can occur when many concurrent requests are processed and provide a simple algorithm for the static analysis of policies that is used to define sufficient constraints on the concurrency of the enforcement mechanism to avoid conflicts.

We base our discussion on the well-known UCON model [1] that presents an extension of more traditional access control, allowing for ongoing control of long standing interactions. We show how the specification of a single usage process can be structured to clearly separate the user, the controller and the system in the design. We model the controller using Statecharts [2] and provide a semantics in Interval Temporal Logic [3].

In Section II we give a short introduction to Interval Temporal Logic and its use in the paper to formalise the presented Statecharts and additional constraints on the behaviour of the enforcement mechanism that represent policies. In Section III we outline the UCON model and provide a Statechart representation that distinguishes the user, controller and the system as orthogonal components. In Section IV we formalise a simplified controller for stateless policies and build upon this to show how stateful UCON policies can be enforced. We show that a stateful policy introduces potential conflicts between concurrently controlled usage processes using a small example. In Section V we present a straight-forward solution that processes all requests interleaved and is thus avoiding conflicts. We discuss the benefits and drawbacks of this solution w.r.t. large-scale distributed systems. In Section VI we then present a simple algorithm that statically analyses the constraints on the controller imposed by the policy and determines potential conflicts. The resulting dependency graph is then used to constrain the behaviour of the initially presented controller such that conflicts are avoided. In Section VII we review related work. We conclude the findings of this paper in Section VIII and outline future work in this area.

II. Preliminaries

ITL is used in this work for two reasons, firstly it allows us to express accurately the behaviour of the controller w.r.t. events, secondly we use ITL to constrain the behaviour of the controller and show how UCON policies can be expressed as such constraints. Temporal logic is a well suited formalism to express behaviours of a system. The reason for choosing ITL over other temporal logic formalisms is the integration of our previous work on the refinement of history-based policies and its compositionality that is evident in the specification of the presented Statecharts.

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0,\ \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of integer values $\mathbb{Z}$. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

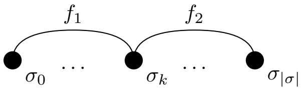
The syntax of ITL is defined in Figure 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

	Expressions
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
	Formulae
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$ $\mathsf{skip} \mid f_1 \ ; f_2 \mid f^*$

1: Syntax of ITL

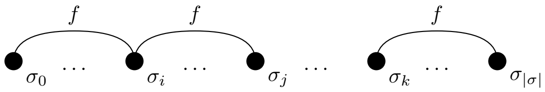
The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 \; ; \; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Figure 2.



2: Informal Semantics of $f_1 \; ; f_2$

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Figure 3.



3: Informal Semantics of f^*

- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- $\mathsf{fin}\ v$: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

A. Derived Constructs

The following is a list of some derived constructs that are used in the remainder of this paper.

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \; ; f$ next f, f holds from the next state. Example: $\bigcirc(X = 1)$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.

$\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$ non-empty interval, i.e., any interval of length at least one.

$\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$ interval, i.e., any interval of length zero (just one state).

$\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true} \; ; \mathsf{false}$ infinite interval, i.e., any interval of infinite length.

$\mathsf{finite} \mathrel{\widehat{=}} \neg\,\mathsf{inf}$ finite interval, i.e., any interval of finite length.

$\Diamond f \mathrel{\widehat{=}} \mathsf{finite} \; ; f$ sometimes f, i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ always f, i.e., any interval such that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.

$\Diamond_{\mathrm{i}}\, f \mathrel{\widehat{=}} f \; ; \mathsf{true}$ diamond-i, i.e., any interval such that f holds over a prefix sub-interval.

$\boxed{\mathrm{i}}\ f \mathrel{\widehat{=}} \neg \Diamond_{\mathrm{i}} \neg f$ box-i, i.e., any interval such that f holds over all prefix sub-intervals.

$\Diamond_{\mathrm{a}}\, f \mathrel{\widehat{=}} \Diamond(\Diamond_{\mathrm{i}}\, f)$ diamond-a, i.e., any interval such that f holds over a sub-interval.

$\boxed{\mathrm{a}}\ f \mathrel{\widehat{=}} \neg \Diamond_{\mathrm{a}}(\neg f)$ box-a, i.e., any interval such that f holds over all sub-intervals.

$\mathsf{keep}\ f \mathrel{\widehat{=}} \boxed{\mathrm{a}}\ (\mathsf{skip} \supset f)$ keep f, i.e., any interval such that f holds over all unit sub-intervals.

$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$ final state, i.e., any interval such that f holds in the final state of that interval.

$v := e \mathrel{\widehat{=}} (\bigcirc v) = e$ assignment, i.e., the value of v will be e in the next state.

$v \leftarrow e \mathrel{\widehat{=}} \mathsf{finite} \wedge (\mathsf{fin}\ v) = e$ temporal assignment, i.e., the value of v in the final state will be the value of e.

$\mathsf{stable}\ v \mathrel{\widehat{=}} \Box(\mathsf{more} \supset v := v)$ remain stable, i.e., the value of v remains stable in the interval.

B. State Semantics

The state of an independent system component c is modelled by the state variable S_c that can assume any of the distinct constants $x \in states_c$ in the set of states of that component. Each event is modelled as a Boolean state variable E_{event}. Generating an event means setting $E_{event} = \mathsf{true}$. α and ω denote sets of these event variables; α denotes the events (or event-expressions) that label a transition leading to the state and ω the denotes the events that label a transition leaving the state.

$$\begin{aligned}\varphi_c(x, \alpha, \omega) \quad \widehat{=} \quad & S_c := x \wedge (\bigcirc \mathsf{stable}\,(S_c)) \wedge \mathsf{more} \wedge \\ & (\bigvee_{v \in \alpha} v) \wedge \mathsf{keep}\,(\neg \bigvee_{v \in \omega} v) \wedge \mathsf{fin}\,(\bigvee_{v \in \omega} v)\end{aligned}$$

The transition from one state to another is not instantaneous, but takes exactly a unit interval. The variable S_c, indicating the state is maintained throughout the whole interval. A state is at least a unit-interval in length. In the beginning of the interval describing the state, one of the event variables in α, that label the transitions leading to this state, is true. Throughout the interval none of the events that label transitions leaving the state is true, only in the final state at least one of them is true.

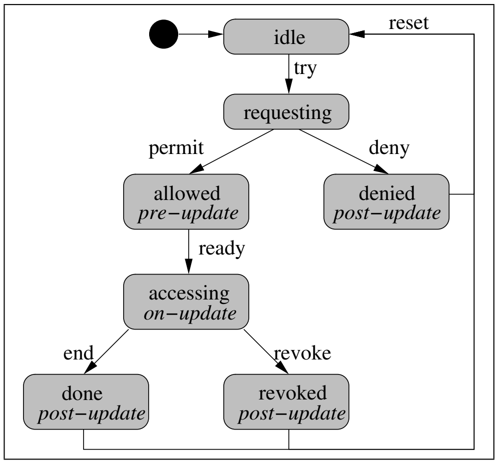
This obviously does not represent a full formalisation of Statecharts in ITL, however it is simple and expressive enough to discuss the concurrency of mechanisms enforcing usage control policies.

III. Usage Control

Usage control (UCON) [1] has been described as a new paradigm for access-control, that extends conventional access checks with ongoing control and management functionality. In UCON an access is not seen as an atomic access request, but as a *usage process*, viz. a longstanding interaction with a system resource.

A. Usage Processes

A usage process describes a partial behaviour of the system when an authenticated user is accessing a system resource. A usage process can be in various states as depicted in Fig. 4.



4: Single Usage Process

Initially the usage process is in the *idle* state. The user will at some point *try* to access the resource and change the state to *requesting*. Now the UCON monitor will decide, based on the policy, whether to *permit* or *deny* the access, changing the state to either *allowed* or *denied*, respectively. In the state *denied*, the UCON monitor will perform postupdate actions as defined by the policy and on completion reset the state of the usage process to *idle*. In the state *allowed* the UCON monitor will perform preupdate actions that are specified in the policy for this request and on completion transit into the state accessing that represents the actual interaction between the user and the resource. During the access, the UCON monitor will perform onupdate actions at certain points during the execution. The access is then either *end*ed by the user or *revoked* by the UCON monitor if a revocation condition specified in the policy has been met. In the state *done* the request has been executed successfully. The state *revoked* indicates the failure, due to policy violation. In either case the *postupdate* actions specified in the policy are executed by the UCON monitor and the usage process is reset to its *idle* state.

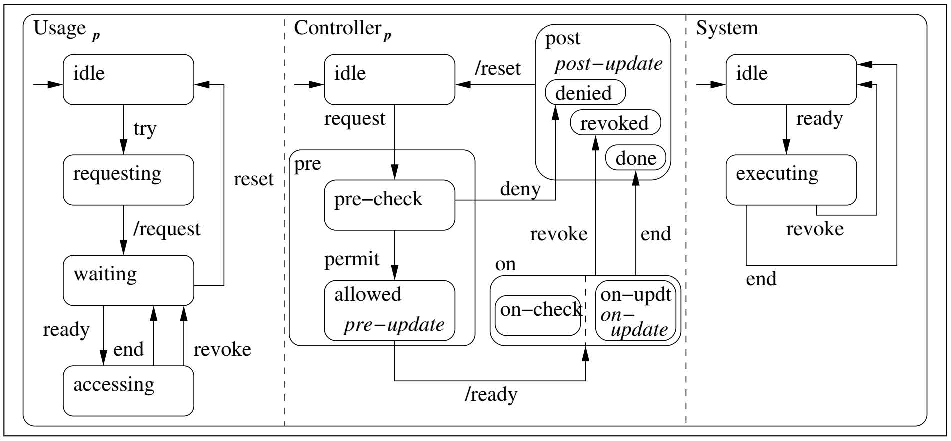
This is a variation of the UCON model originally described in [4]. The difference is that pre-update actions are executed in the *allowed* state rather than in the *requesting* state, viz. they cannot affect the access-control decision. For this purpose the state *allowed* has been introduced. Another difference is that post-update actions are executed at the end of a request, irrespective of whether it was *denied* or *revoked* by the controller or *end*ed by the user. As we are interested in the concurrency of usage processes, we take the view that the same usage process can be performed several times and reset it after every request in its *idle* state.

B. User, Control and System Separation

The depiction of a usage process in Fig. 4 explains the behaviour of the system from the viewpoint of a single user request, however it lacks structure. It does not distinguish clearly between the user, the controller enforcing the policy, and the system that provides the actual functionality requested by the user. This structure is important if we consider the enforcement of usage control in a real system. Fig. 5 models usage control as a reactive system in form of a Statechart. Here the usage process, its controller and the system itself are modelled as orthogonal components that are synchronised using broadcast events. The user will initiate a usage process p by generating the *try* event. In the *requesting* state all required activities for the usage process to start are executed and the *request* event is generated. Upon the *request* event the controller for the usage process p will enter the state *pre-check* and determine the access control decision based on the policy. If the access is denied it will perform the post-update activities in the state *post* and indicate the resetting of the usage process by generating the event *reset*. If the access was permitted the pre-update activities are performed in the state *pre* and upon completion the event *ready* is generated to indicate that the access can commence. All three components synchronise on this event, viz. the states *accessing*, *on* and *executing* are entered simultaneously. In the *on* state the controller performs on-update activities as defined by the policy and can also raise the event *revoke* that terminates the access. Refinements of the states *accessing* and *executing* describe the interaction between user and system during the access. The event *end* indicates the termination of the usage by the user.

The usage and control components are parametrised with p, denoting a concrete usage process. A usage process represents a subject performing and action on a resource. All usage processes are independent from each other. In the model Fig. 5 also the controllers are independent from each other.

Whilst it is desirable to have independent controllers, as this increases the concurrency of the policy enforcement, we will show in the following that policies introduce dependencies between controllers that require their synchronisation. When policies define *mutable attributes* and their updates, then these attributes are *shared* between the various controllers of usage processes. This shared state allows for side-effects to take place across usage processes. A simple example is the Chinese Wall policy [5] where the use of one resource determines future access control decisions for other accesses. A Chinese Wall between a set of usage processes or Separation of Duty requirements would be achieved through updating a shared state and making access and revocation conditions dependent on this state.



5: User, Controller and System

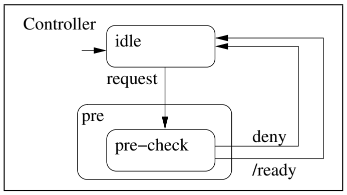
The existence of a shared state requires the synchronisation of the various controllers, to avoid concurrent write situations. However it is inherently inefficient to have all controllers operating interleaved to ensure that no conflicts occur. Instead it is much more sensible to analyse the policy for such dependencies and constrain the behaviour of the controllers such that conflicts are avoided. In the following we show how controller for stateless UCON policies ($preA_0$) can be designed, we then extend this to include UCON policies that perform pre-updates of mutable attributes ($preA_1$). We show how simple exclusion requirements can be problematic if the controllers do not take care of the dependencies between usage processes.

IV. Enforcing UCON policies

Traditional access control considers a single controller that determines the access decision. In the most basic case all access decisions are static, e.g. defined in Access Control Lists, and the controller will not maintain any state (Fig. 6). In this case no conflicts can arise and all access decisions can be made independently. However, as soon as the controller is extended to enforce stateful policies, viz. dynamically updates attributes as side-effects of access requests (Fig. 7), the possibility for conflict does arise.

A. Traditional Access Control

Traditional access control corresponds to the UCON $preA_0$ class of policies, viz. an authorisation check is performed before the request is granted and no mutable attributes are updated. This class can be enforced by the controller depicted in Fig. 6. We term here the controller as being the overall enforcing entity, and refer to a reference monitor (RM) as a component of the controller that is responsible for a single usage process p. p stands here for the tuple $\langle s, o, a\rangle$, where s is the subject requesting to perform the action a on resource o. P denotes the set of all usage processes. The states of each RM_p are: $states_{RM_p} \mathrel{\widehat{=}} \{idle_p, check_p\}$. We define the behaviour of the controller as follows:



6: Static Controller

$$\varphi_{idle,p} \mathrel{\widehat{=}} \varphi_{RM_p}(idle, \{E_{init_p}, E_{deny_p}, E_{rdy_p}\}, \{E_{req_p}\}) \quad (1)$$

$$\varphi_{check,p} \mathrel{\widehat{=}} \varphi_{RM_p}(check, \{E_{req\,p}\}, \{E_{rdy\,p}, E_{deny\,p}\}) \quad (2)$$

$$\varphi_{RM_p} \mathrel{\widehat{=}} S_{RM_p} = idle \wedge (\varphi_{idle,p}\,;\varphi_{check,p})^* \quad (3)$$

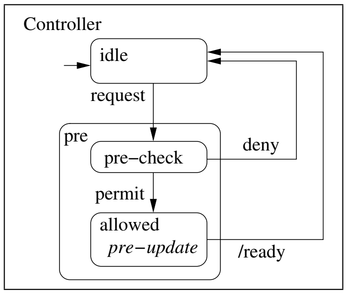
$$\varphi_{control} \mathrel{\widehat{=}} \bigwedge_{p\in P} (\varphi_{RM_p}) \quad (4)$$

(Eq. 1) defines the behaviour of the controller in the state $idle_p$. The state can be entered by a transition labelled with the $init_p$, $deny_p$ or rdy_p event and left with transitions that are labelled with the req_p event. Analogously the state $check_p$ is defined in (Eq. 2). The behaviour of the RM (Eq. 3) for the usage process p is then defined as the alternation between the states $idle_p$ and $check_p$. The overall usage controller is then the conjunction of the RM for all usage processes $p \in P$. Under the assumption that the *idle* and *check* state do not maintain any attributes for future access decisions, all access control checks can indeed executed concurrently (true parallelism) without interference.

An implementation of such an enforcer is e.g. the Java *AccessController* class is accessible by all threads running in the JVM concurrently. Standard Java policies do not maintain state. Similarly traditional access control lists that are static and are not changed at run-time fall into this category.

B. Stateful Access Control

The notion of mutable attributes as proposed by Park et.al. [6] provides far reaching flexibility for access control specifications. Attributes can change their values as a *side-effect* of accesses that are initiated by the user. This means that the reference monitor is stateful and maintains attributes that influence future control decisions. Using mutable attributes for example history-based policies can be expressed, or the number of users concurrently accessing a resource can be limited. With respect to UCON the simplest form of using mutable attributes are policies that are expressed as $preA_1$, viz. the attribute update is performed before the access takes place. The controller depicted in Fig. 7



7: Dynamic Controller

performs update action prior to the access and can therefore enforce this class of policies. The states of the RM are $states_{RM_p} \mathrel{\widehat{=}} \{idle, check, allowed\}$. The behaviour of a controller is then defined as:

$$\begin{aligned}
\varphi_{idle,p} &\mathrel{\widehat{=}} \varphi_{RM_p}(idle, \{E_{init_p}, E_{deny_p}, E_{rdy_p}\}, \{E_{req_p}\}) && (5)\\
\varphi_{check,p} &\mathrel{\widehat{=}} \varphi_{RM_p}(check, \{E_{req_p}\}, \{E_{perm_p}, E_{deny_p}\}) && (6)\\
\varphi_{allowed,p} &\mathrel{\widehat{=}} \varphi_{RM_p}(allowed, \{E_{perm_p}\}, \{E_{rdy_p}\}) && (7)\\
\varphi_{RM_p} &\mathrel{\widehat{=}} S_{RM_p} = idle \wedge (\varphi_{idle,p} \,;\, \varphi_{check,p} \,;\\
&\qquad (\varphi_{allowed,p} \oplus (\mathsf{empty} \wedge E_{deny_p})))^* && (8)\\
\varphi_{control} &\mathrel{\widehat{=}} \bigwedge_{p \in P} (\varphi_{RM_p}) && (9)
\end{aligned}$$

The state *allowed* does explicitly perform *pre-update* actions to modify the mutable attributes of the policy. These update actions are part of UCON policies and are defined as a constraint on the state *allowed*. For example the increment of an access counter $count_p$ every time an access takes place can be expressed as:

$$\boxed{\text{a}}\ (\varphi_{allowed,p} \supset (count_p \leftarrow count_p + 1))$$

Informally this states that every time the reference monitor for the usage process p is in the state *allowed* the mutable attribute $count_p$ is incremented. The execution of the $allowed_p$ state implies that the value of $count_p$ at the end of the state is one more than in the beginning of the state $allowed_p$. In this paper we take the view that an attribute update can be expressed as the assignment of the a set of attributes to new values. More complex update activities can be introduced, which however complicates the model and its analysis. Attributes can be shared between the reference monitors for the various usage processes. Dependencies can be imposed by a policy, that make the parallel enforcement (Eq. 9) unfeasible. We present an example of such a dependency in the following.

C. Example of Conflicting Attribute Update

Suppose the following requirement:

> *Users alice* or *bob can read* from *file foo*, but never both, viz. after bob accessed the file *alice* can no longer read it and vice versa.

To model this requirement we introduce a mutable attribute *foo.readby* that stores the identity of the subject that has access to the file *foo*. Initially *foo.readby* is set to *unknown*. The requirement is then captured by the pre-update rules:

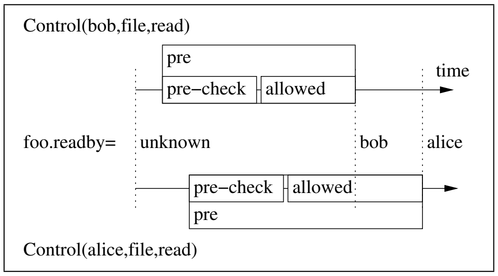
$$\begin{aligned}
\boxed{\text{a}}\ \varphi_{allowed,\langle bob,foo,read\rangle} &\supset \mathit{foo.readby} \leftarrow bob\\
\boxed{\text{a}}\ \varphi_{allowed,\langle alice,foo,read\rangle} &\supset \mathit{foo.readby} \leftarrow alice
\end{aligned}$$

The *preupdate* rules constrain the behaviour of the controller in the *allowed* state. The controller must store the identity of the accessing subject in the attribute *foo.readby*. It is important to note that the two reference monitors for *alice* and *bob* are executing concurrently.

Authorisation rules are enforced by constraining the choice of the *permit* respectively *deny* events that are raised by the controller. We describe the authorisations conditions for both usage processes as the following invariants:

$$\begin{aligned}
\Box(E_{permit_{\langle bob,foo,read\rangle}} &\supset \mathit{foo.readby} = bob \vee\\
&\qquad \mathit{foo.readby} = unknown)\\
\Box(E_{permit_{\langle alice,foo,read\rangle}} &\supset \mathit{foo.readby} = alice \vee\\
&\qquad \mathit{foo.readby} = unknown)\\
\Box(E_{permit_p} &\supset \neg E_{deny_p})\\
\Box(E_{deny_p} &\supset \neg E_{permit_p})
\end{aligned}$$

These rules state that the *permit* event can only occur if either no user has read the file *foo* yet, or the file has previously been read by the same user. The two reference monitors process their access control decisions and update actions concurrently. As the attribute $foo.readby$ is shared between the two processes a concurrent write can occur. The effect of this is that the requirement is not enforced by the controller if the requests are made in very short succession. Figure 8 depicts this concurrent update.



8: Concurrent Requests

The access control check for both usage processes succeeds, as *foo.readby* has at the time the value *unknown*. The *preupdate* for *bob*'s usage assigns the value *bob* that is immediately overwritten by *alice*. The effect is that both *bob* and *alice* can read the file *foo*, and only *alice* will have access in the future. If *alice*'s request was made a little later, she would have been denied the access, as then the value of *foo.readby* would be already *bob*. This shows that the policy definition that models the exclusive access of *alice* and *bob* introduces a dependency that must prevent the access control requests to be processed concurrently. The *pre* states of the controllers must be mutually exclusive in order to prevent undesired concurrent modifications of shared attributes.

V. Avoiding conflicts by interleaving

A relatively straightforward solution for a $preA_1$ controller is to interleave all access decisions and pre-update actions of its reference monitors. The overall behaviour of such a controller can then be expressed as a single process:

$$\begin{aligned}\varphi_{control} \;\widehat{=}\; & S_{RM} = idle \wedge \\ & (\varphi_{init} \;; \\ & \bigoplus_{p \in P}(\varphi_{check,p} \;; (\varphi_{allowed,p} \oplus (\mathsf{empty} \wedge E_{deny_p}))))^{*}\end{aligned} \tag{10}$$

The operator $\oplus$ stands here for the logical exclusive-or. The controller starts in the state *idle* and upon the event *request* enters the state *pre* of *exactly one* usage process p. All states *pre*, viz. the sequence of *check* and optionally *allowed*, are mutual exclusive.

1) Advantage: The advantage of this enforcement model is that conflicting concurrent updates are effectively avoided and that the approach is independent from the enforced policy. A concrete implementation of such controller, this could be easily programmed using monitors or binary semaphores. The implementation is generic and is not dependent on the concrete policy that is enforced. Most approaches in which the policy is interpreted will make this or similar design choices.

2) Disadvantage: The drawback is that the interleaving of all usage processes introduces an unnecessary bottleneck. This is especially serious when the controller is extended to deal with ongoing updates of attributes and revocation that are part of more complex UCON policies (e.g. $preA_2$ or onA_* models). An interleaving enforcement model also is difficult to distribute over a network as it requires additional protocols (e.g. 2-phase commit) to ensure that updates are not made concurrently. In a distributed enforcement model the attributes are typically stored in a Policy Information Point (PIP) [7], [8] and are accessed by the various decision points in the network. The overhead on communication and synchronisation with the PIP can be significant.

3) Discussion: The interleaving enforcement model assumes that the decision making in the state *check* and updates in the state *allowed* are sufficiently fast. With respect to the example the processing of *alice's* request would be delayed until *bob's* request has been processed, resulting in a negative access control decision for *alice*. This enforcement design does allow to capture the requirement of exclusive access between *bob* and *alice*. This model is relatively easy to implement using e.g. monitors to mark the critical sections in the enforcement code. The approach is adequate for relatively small systems, or systems where requests are not processed concurrently anyway. If the underlying system however is capable of processing a number of requests, the interleaving controller introduces a bottleneck.

VI. Efficient Enforcement Mechanism Design

The degree of concurrency that can be provided by a controller depends on the type of policy that is enforced. We have shown in Section IV a controller for stateless policies that can unconditionally process requests concurrently. That this is not in general possible for stateful policies is clear from the example given in Section V. Fortunately it is not only the type of policy that determines the degree of concurrency that is possible in the enforcement of a policy, but the rules themselves.

Suppose that two other users, *chris* and *john*, are unconditionally allowed to *read* from the file *foo*. In the case of the interleaving enforcement all access request would be processed sequentially, leading to an increased delay. In fact, there is no need to delay requests by *chris* and *john* as they are fully independent. If the system is operating under a high load, viz. many requests are made concurrently, the time-span between the user request and the access decision can be substantially decreased by controlling the usage processes for *chris* and *john* concurrently. For this extended example the controller would control four usage processes, representing the access to the file *foo* by *alice*, *bob*, *chris* and *john*. The constraints for the controller are as follows:

$\langle alice, foo, read\rangle$:

$$\begin{aligned}\boxed{\text{a}}\;\varphi_{allowed,\langle alice,foo,read\rangle} &\supset foo.readby \leftarrow alice \\ \Box(E_{permit_{\langle alice,foo,read\rangle}} &\supset foo.readby = alice \vee \\ & \quad foo.readby = unknown)\end{aligned}$$

$\langle bob, foo, read\rangle$:

$$\begin{aligned}\boxed{\text{a}}\;\varphi_{allowed,\langle bob,foo,read\rangle} &\supset foo.readby \leftarrow bob \\ \Box(E_{permit_{\langle bob,foo,read\rangle}} &\supset foo.readby = bob \vee \\ & \quad foo.readby = unknown)\end{aligned}$$

$\langle chris, foo, read\rangle : \quad true$
$\langle john, foo, read\rangle : \quad true$

By breaking down the policy into the constraints of the controller for the individual usage processes, the dependencies between usage processes can be statically analysed.

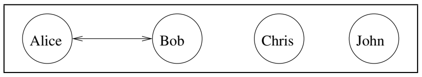
A. Identification of Dependencies

In order to determine the dependencies between the usage processes the mutable attributes that occur in constraints on the *allowed* state and the constraints on access decisions must be compared. This is a simple algorithm that traverses the potentially conflicting usage processes and determines whether a conflict can occur or not. Here P is the set of all usage processes and *Var*(x,y,z) denotes the mutable attributes that are used in constraints for state x and usage process y; z indicates whether access to the variable is read (r) or write (w).

```
for each  $p \in P$ :
  for each  $q \in P \setminus \{p\}$ :
    if $Var$(pre-check, p, r) $\cap$ $Var$(allowed, q, w) $\neq \emptyset$ then
      Decision depends on update.
      Pre-states of $p$ and $q$ are mutually exclusive
    endif
    if $Var$(allowed, p, w) $\cap$ $Var$(allowed, q, w) $\neq \emptyset$ then
      Potential for conflicting updates of attributes.
      Pre-states of $p$ and $q$ are mutually exclusive
    endif
    if $Var$(allowed, p, r) $\cap$ $Var$(allowed, q, w) $\neq \emptyset$ then
      Pre-states of $p$ and $q$ are mutually exclusive
    endif
    if $Var$(allowed, p, w) $\cap$ $Var$(allowed, q, r) $\neq \emptyset$ then
      Pre-states of $p$ and $q$ are mutually exclusive
    endif
  endfor
endfor
```

If the constraints for the two usage processes share mutable attributes they are dependent. The first case is that one access control decision relies on attributes that are updated in the by another usage process' pre-update. The second case is that two processes can try to update the same attributes concurrently. The last two cases capture that one usage process' update relies on values that are updated by another usage process.

Applying this algorithm to our example policy for *alice, bob, chris* and *john* results in the mutual exclusion of the processes for *alice* and *bob*. Only the first and the second case apply. The result can be visualised as a dependency graph:



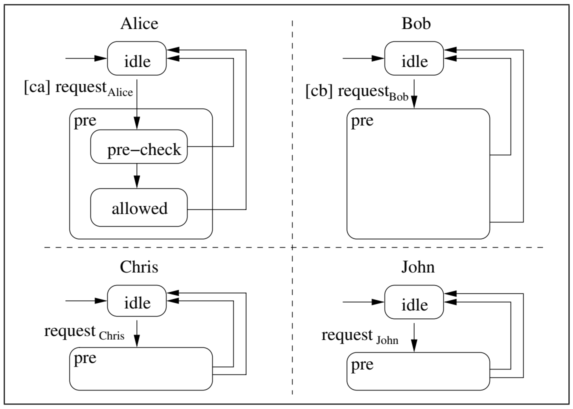
9: Dependency Graph

Nodes represent usage processes. Edges represent dependencies. Neighbouring nodes in the graph are mutually exclusive. Independent sub-graphs, e.g. {alice,bob}, {chris}, {john} can be enforced without any interaction between the controllers. This is advantageous for the distribution of controllers, as they can operate without any communication. Nodes that are connected, but not direct neighbours, can be processed concurrently, but must remain on the same controller to guarantee mutual exclusion with other usage processes.

a) Remark: The presented algorithm is based on the enforcement of UCON policies of type $preA_1$ (See Fig. 7). Due to space limitations we only outline the extensions required for the full set of UCON policies (See Fig. 5). The extension requires more cases, checking for attribute conflicts, to be added. For $preA_1$ it is sufficient to address the mutual exclusion of the usage process' *pre* states. In the extension the conflicts must distinguish between the states *pre*, *on* and *post* and attach these as labels on the edges of the dependency graph. This results in a graph that shows which states between two dependent usage processes must be mutually exclusive. This is important as for the control of long-standing interactions between users and system any interleaving of whole usage processes is highly undesirable.

B. Constraining Dependent Controllers

Having identified the dependencies between the controllers of the usage processes we need to ensure mutual exclusion of the affected states. Figure 10 shows the four reference monitors for the $preA_1$ controller of our example. The state *pre* is entered on the event *request*. To ensure mutual exclusion we have to further constrain this transition. Conditions in Statecharts are written in brackets preceding the event.



10: Dependency between Controllers

The *request* transitions for the controllers for *Alice* and *Bob* are additionally constrained by the conditions ca and cb, respectively:

$$ca \quad \widehat{=} \quad \neg request_{Bob} \wedge \neg(S_{Bob} = pre - check \vee S_{Bob} = allowed)$$
$$cb \quad \widehat{=} \quad \neg request_{Alice} \wedge \neg(S_{Alice} = pre - check \vee S_{Alice} = allowed)$$

The transition for the usage process of *Alice* to enter the state *pre* is conditional on that the *i)* usage process of *Bob* is not concurrently requesting access and *ii)* usage process of *Bob* not being currently in the state *pre* (which is either *pre-check* or *allowed*). Similar the condition for *Bob*'s usage process. The transitions for *Chris* and *John* are unconditional. As a result the *pre* states for *Alice* and *Bob* are mutually exclusive. The access checks and pre-updates for *Alice* and *Bob* are interleaved and those for *Chris* and *John* are truly concurrent.

VII. Related Work

The UCON model has been initially proposed by Park and Sandhu in [1]. A formalisation has been presented by Zhang et.al [4]. In our previous work [9] we provided an alternative formalisation focusing on implementation issues. We selected UCON as a suitable abstraction level for the refinement of high-level history-based policies [10] into concrete implementable enforcement code that we presented in [11].

Work on the analysis of policy conflicts has been undertaken in the past, e.g. Lupu and Sloman [12] focus on the analysis of conflicts within policies that result of conflicting specification, e.g. the case that an obligation and a negative authorisation can be derived simultaneously. Work by Jajodia et.al. [13]

addresses the resolution of conflict by means of decision rules that decide at run-time whenever conflicts arise in the policy. Similarly Chomicki and Lobo [14] introduce monitors for history-based policies based on the PDL framework [15]. Their work is possibly closest related as it addresses the deconfliction of *actions* by means of delaying or cancelling actions. Their work is focused on the run-time aspect of policy enforcement and does not address the static analysis as a means to optimise the enforcement efficiency.

Schneider [16] proposed an automata-based approached to the enforcement of history-based policies. He categorises properties that are enforceable by execution monitoring. His work has been later extended by [17] to include the enforcement of obligations. More recently a classification has been presented by Hamlen et.al. [18]. Schneider also allows for the conjunction of security automata however does not address the identification of dependencies from higher-level policies.

VIII. Conclusion

We initially presented a model of an enforcement mechanism as a reactive system in form of a Statechart. This model clearly separates between user, controller and system. We focused on the controller component as UCON policies can be expressed as constraints on the controller's behaviour.

We formalised subsets of the controller for the UCON policy classes $preA_0$ and $preA_1$ and showed that the introduction of stateful policies, in the case of UCON represented by mutable attributes, introduce dependencies between the controllers for usage processes. We motivated the resulting requirement for the interleaving of usage processes control states and provided an example of how an update conflict can invalidate the original requirement.

In Section V we discussed the option of interleaving all usage processes and found that this constitutes a significant bottleneck if the system is *i)* able to concurrently process user requests and *ii)* expecting substantial load. For such systems we proposed to statically analyse the policy before deployment to identify synchronisation requirements of the controller and provided a simple algorithm to identify dependencies based on the attributes that are used in the policy rules. Based on the identified dependencies we then showed how the enforcement model can be constrained.

The proposed approach is especially useful for large-scale distributed environments where many user requests are made concurrently, and also to increase the decision making time in the case that the controller is running on increasingly common multi-processor systems.

We see this work as the basis to further analyse dependencies from policies. For example we did not consider that update rules are typically conditional, viz. when our algorithm detects a conflict it may well be that the conditions of the rules ensure that this conflict cannot occur at run-time. The application of model-checking to identify these cases on a rule-by-rule basis is a future strand of investigation. Another aspect is the distribution of controllers within a distributed system. Having said that the distribution based on independent subgraphs is straight forward, it is of possibly even greater interest to identify distributions of controllers that minimise communication overhead, whilst maximising concurrency. This requires a detailed analysis at attribute level. A third item for our future work is the integration of the enforcement model with our previous work on policy specification and refinement.

References

[1] J. Park and R. S. Sandhu, "The UCON$_{ABC}$ usage control model." *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 128–174, 2004.

[2] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987. [Online]. Available: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

[3] A. Cau, B. Moszkowski, and H. Zedan, "The ITL homepage," http://www.cse.dmu.ac.uk/STRL/ITL.

[4] X. Zhang, F. Parisi-Presicce, R. S. Sandhu, and J. Park, "Formal model and policy specification of usage control." *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 4, pp. 351–387, 2005.

[5] D. Brewer and M. Nash, "The Chinese Wall Policy," in *IEEE Symposium on Research in Security and Privacy*. Oakland, California, USA: IEEE, May 1989, pp. 206–214.

[6] J. Park, X. Zhang, and R. S. Sandhu, "Attribute Mutability in Usage Control." in *Proceedings of IFIP TC11/WG 11.3 Eighteenth Annual Conference on Data and Applications Security*, C. Farkas and P. Samarati, Eds. Sitges, Catalonia, Spain: Kluwer, July 2004, pp. 15–29.

[7] ISO/IEC, "ISO/IEC 10181-3:1996 Information technology – Open Systems Interconnection – Security frameworks for open systems: Access control framework," March 2006. [Online]. Available: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=18199

[8] OASIS, "eXtensible Access Control Markup Language (XACML) Version 2.0," February 2005. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20

[9] H. Janicke, A. Cau, F. Siewe, and H. Zedan, "A note on the formalisation of UCON," in *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT07)*, June 2007, pp. 163–168.

[10] H. Janicke, A. Cau, F. Siewe, H. Zedan, and K. Jones, "A Compositional Event & Time-based Policy Model," in *Proceedings of POLICY2006, London, Ontario, Canada*. London, Ontario Canada: IEEE Computer Society, June 2006, pp. 173–182.

[11] H. Janicke, A. Cau, F. Siewe, and H. Zedan, "Deriving Enforcement Mechanisms from Policies," in *Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007)*, June 2007, pp. 161–170.

[12] E. C. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Trans. Softw. Eng.*, vol. 25, no. 6, pp. 852–869, 1999.

[13] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian, "Flexible support for multiple access control policies," *ACM Trans. Database Syst.*, vol. 26, no. 2, pp. 214–260, 2001.

[14] J. Chomicki and J. Lobo, "Monitors for History-Based Policies," in *Proceedings of the international Workshop on Policies For Distributed Systems and Networks*, ser. Lecture Notes In Computer Science, J. L. M. Sloman and E. Lupu, Eds., vol. 1995. Springer-Verlag, January 2001, pp. 57–72. [Online]. Available: http://www.cse.buffalo.edu/~chomicki/papers-policy01.ps

[15] J. Lobo, R. Bhatia, and S. A. Naqvi, "A Policy Description Language," in *AAAI/IAAI*, 1999, pp. 291–298. [Online]. Available: http://citeseer.ist.psu.edu/lobo99policy.html

[16] F. B. Schneider, "Enforceable Security Policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, February 2000.

[17] C. N. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, "Enforcing Obligations with Security Monitors," in *The Third International Conference on Information and Communication Security (ICICS'2001)*, November 2001.

[18] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 175–205, 2006.